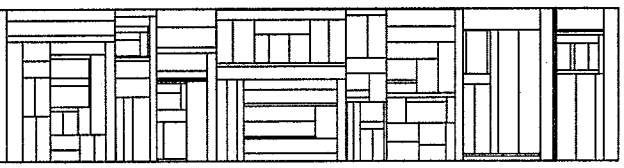
ON TYPE DEFINITIONS WITH PARAMETERS

by

Marvin Solomon

DAIMI PB-54
November 1975

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55



On Type Definitions With Parameters

Table of Contents

On Type Definitions With Parameters

by Marvin Solomon
Department of Computer Science
Aarhus University
Aarhus, Denmark
Phone : 06-128355

Abstract

This paper analyzes some of the consequences of allowing the definition of parameterized data types in programming languages. A typical use of such types is :

type queue (x) = struct (x, ref (queue (x))),

intqueue = queue (int).

It is shown that the addition of parameters permits the definition of new types not definable without parameters. In particular, the types definable with parameters are closely related to the deterministic context-free languages, whereas the author has previously shown that the types definable without parameters are characterized by the regular (i. e. finite state) languages.

An important consequence of this fact is that the type equivalence problem, which is easily solvable in the absence of parameters, becomes equivalent to the (currently open) equivalence problem for deterministic pushdown automata.

Keywords and phrases : data types, modes, modals, lattice-theoretic models, deterministic pushdown automata, formal semantics.

CR Categories : 4. 12, 5. 22, 5. 23, 5. 27.

1. INTRODUCTION

Several authors [4, 6, 8, 12] have pointed out the usefulness of parameterized data types in a highly typed language such as ALGOL 68 [18] or Pascal [9]. For example, a general queueing facility might be defined by

(1.1) type queue(x) = struct (x, ref queue (x)) and later used by including definitions such as

(1.2) type intlist = queue (int)

(1.3) type waiting line = queue (person)

(where person is some programmer defined data type). We shall adopt the terminology of [8, 12] and call an object like queue a modal.

It should come as no great surprise that allowing modals creates new difficulties for the implementor. For example, if a parameter to a procedure can be declared to be of type queue (x), then the code generated may depend heavily on the size of objects of type x, and even if x is passed as a parameter (of type type) a good deal of run-time checking may be needed. Lindsey [12] suggests allowing only variables of type ref queue (x), and Gehani [4] discusses restrictions under which several copies of the troublesome procedure may be generated.

All these problems arise from attempts to use modals in variable declarations. The purpose of this paper is to point out a more subtle problem that remains even if modals are never used in variable declarations, but only as a tool for defining (ordinary) types as in (1.1-3.). The problem in question is to decide when two definitions are equivalent. As Král [10] points out, the algorithm for equivalence of modes in ALGOL 68 is similar to the equivalence algorithm for finite automata. As we showed in [16], this similarity is due to the fact that

modes defined by ALGOL 68-like mode declarations are regular, in the sense that they are characterized by regular sets of strings. With the introduction of parameters, this no longer holds true; the definable types are now characterized by arbitrarily complex deterministic context-free languages. The result of this is that the equivalence problem becomes of the same level of difficulty as the equivalence problem for deterministic languages. Since the latter problem is at present a famous open problem, we see that the addition of parameters adds an essential new complication.

1.1 Equivalence of Modes

The foregoing discussion presumes that there is an obvious canonical notion of "equivalence of types". This is not true. One school holds that any two distinct definitions define different types. Of course in this case the equivalence problem is trivial. We will take a point of view closer to that of ALGOL 68, one which has been supported in [11] and [16]. Briefly, we say that two types are equivalent if all values of one type are "the same shape as" values of the other type. To be somewhat more specific, we assume that there are atomic values partitioned into disjoint atomic type classes, and structuring operators which construct values from component values. Two values are the same type if they are both of the same atomic type or if they are constructed by the same operator from components which are, respectively, of the same types. This defines (recursively) an equivalence relation: "to be the same type as", and types are nothing more than equivalence classes. The important point is that the type of a variable x should tell the following about the value of x: either the type of x is atomic; then the value of x is atomic and of that atomic type or the type of x is $f(t_1, \ldots, t_k)$ where f is a structuring

operator and $t_1, \ldots, t_k$ are types; then k selection operations are applicable to the value of x and the result of applying the i'th is a value of type t_i. By induction, then, the type of a variable is sufficient to determine whether any given finite sequence of selection operations is applicable to x, and if so, what the type of the result will be. This means that the type of information may be summarized as a (possibly infinite) tree.

1.2 Example

Figure 1a gives an example of a type definition which defines a modal of one argument and applies it to a constant type to obtain a new type. Figure 1b "unrolls" the definition by repeatedly replacing occurrences of the variable v_0 by its definition. The result of applying v_0 to int is shown in Figure 1c. As we shall see later, this type could not be defined without parameters.

The contents of the remainder of this paper are as follows: In section 2, we formally present a syntax for a type definition facility with parameters so as to have a concrete example to illustrate our ideas. In section 3, we endow the facility with a formal semantics based on ideas from [11, 14, 15, 16] and elsewhere. The reader who is satisfied that the trees of figures 1b and 1c accurately reflect the "meaning" of the definition is figure 1a may skip this section, at least on first reading. He should, however, look at those paragraphs labeled "notation". Section 4 contains the central results of this paper (4.2.4 and 4.3.7). We show that types defined in our example language fragment are "deterministic context-free" and hence that the equivalence problem for types is of equal difficulty as the (open) equivalence problem for deterministic languages. Section 5 contains a discussion

of these results and their implications for language design.

2. A TYPE DEFINITION SYSTEM

Our example language fragment will be a stripped-down version of the type declaration facilities of ALGOL 68 [18] or Pascal [9] minimally augmented with the ability to specify a partially defined type as in (1.1) or figure 1a. We assume that the following lexical classes are defined:

- Ω, a set of modal constants. Each $\sigma \in \Omega$ has a rank (number of arguments) which is a non-negative integer. Modal constants of rank 0 may be called type constants and correspond to atomic types such as int or real. Modal constants of rank > 0 are the built-in type constructors of the language, such as struct and ref of ALGOL 68.

- V, a set of modal variables. These also have ranks associated with them. A variable of rank 0 may be thought of as a type variable. Thus ordinary (non-parameterized) types are a special case of modals. For example, in figure 1a, $v_0, v_1 \in V$ and rank $(v_0) = 1$, rank $(v_1) = 0$.

- X, a set of parameter symbols. These are all of rank 0. We now specify the syntax of a modal definition using BNF and some context-sensitive restrictions.

  <definition> ::= <declaration> | <definition><declaration>

  <declaration> ::= <modal variable> <formal parameters>
  = <modal expression>

<formal parameters> ::= <empty> | (<parameter list>)

<parameter list> ::= <parameter symbol> | <parameter list>, <parameter symbol>

<modal expression> ::= <modal constant> <actual parameters>
| <modal variable> <actual parameters>
| <parameter symbol>

<actual parameters> ::= <empty> | (<expression list>)

<expression list> ::= <modal expression>
| <expression list>, <modal expression>

Restrictions

(1) There is exactly one <declaration> for each <modal variable> appearing in the <definition>.

(2) If $\sigma \in \Omega \cup V$ and rank $(\sigma) = 0$, then each <formal parameters> or <actual parameters> following σ is <empty>; if rank $(\sigma) = n$, then each <formal parameters> or <actual parameters> following σ has exactly n members.

(3) Each <parameter symbol> appearing in a <declaration> must appear in the <formal parameters> of that <declaration>. The <parameter symbol>s in any <parameter list> are all distinct.

2.1 Notation

To simplify notation, we will assume for the remainder of this paper that we are talking about one fixed definition $\underline{\underline{D}}$, that the set of modal variables is $V = \{v_0, \ldots, v_{n-1}\}$ for some n, and that $\underline{\underline{D}}$ has the form:

$$
\begin{aligned}
& v_0(x_0, \ldots, x_{\mathrm{rank}(v_0)-1}) = e_0 \\
& \qquad \vdots \\
& v_{n-1}(x_0, \ldots, x_{\mathrm{rank}(v_{n-1})-1}) = e_{n-1}
\end{aligned}
\tag{2.1.1}
$$

where each e_i is an expression. We also adopt a convention due to B. Rosen [13] and omit mention of ranks of symbols when they are evident from the context or irrelevant. For example, $v_0(x_0, \ldots, x_{\mathrm{rank}(v_0)-1})$ is written as $v_0(x_0, \ldots, x_{-1})$. In such a context, "-1" is pronounced "last". □

3. SEMANTICS

Once we have constructed T, the set of types, it should be clear that the meaning of a modal t of rank n is a function fun(t): $T^n \to T$. If each modal constant σ is given a standard meaning fun(σ) and a meaning fun(v_i) is assigned to each modal variable v_i, then each expression e_i represents a function fun(e_i): $T^k \to T$ in an obvious way. (Here k is any integer larger than any j such that x_j appears in e_i.) Since fun(e_i) depends on the assignments of fun(v_j) to v_j, e_i may also be thought of as a functional from $D_0 \times D_1 \times \ldots \times D_{-1}$ to D_i, where D_j is the set of functions from $T^{\mathrm{rank}(v_j)}$ to T. In this way, a definition may be read as a set of simultaneous equations asserting the equality of various functions over T. Under certain circumstances, the equations have a unique solution. (See [16] for more about when the solution is unique. A sufficient condition is that no e_i is of the form $v_j(\ldots)$ or v_j.)

The set we use for T is the same as the one we constructed in [16]. See that paper, [11], and section 1 of this paper for motivation for this choice of model. We briefly review the construction of T and some of its properties.

3.1 Trees and Cpo's

3.1.1 Definition

Let Σ be any ranked set (a set together with a function rank: $\Sigma \to \mathbb{N}$).† A tree over Σ is a partial function $t:\mathbb{N}^* \to \Sigma$ with domain $dom(t) \subseteq \mathbb{N}^*$ satisfying:

(3.1.1) If $\alpha n \in dom(t)$ for some $\alpha \in \mathbb{N}^*$ and $n \in \mathbb{N}$, then $n < rank(t(x))$ and $\beta \in dom(t)$ for all prefixes β of α (including α). □

3.1.2 Notation

Let $T[\Sigma]$ denote the set of trees over Σ. Let $T = T[\Omega]$. We will write $t[\alpha]$ rather than $t(\alpha)$ for the value of t at the string $\alpha \in \mathbb{N}^*$. Let $\bot$ denote the tree with empty domain, and let $t[\alpha] = \bot$ mean that $\alpha \notin dom(t)$. If $\sigma \in \Sigma$, then we also use σ to denote the function $\sigma: T[\Sigma]^{rank(\sigma)} \to T[\Sigma]$ where

---

† $\mathbb{N}$ denotes the set of non-negative integers. $\mathbb{N}^*$ is the set of finite sequences of elements of $\mathbb{N}$ including ϵ, the sequence of length 0.

$$\sigma(t_0,\ldots,t_{-1})[\alpha] = \begin{cases} \sigma \text{ if } \alpha = \epsilon \\ t_i[\beta] \text{ if } \alpha = i\beta \text{ and } i < \text{rank}(\sigma) \\ \text{undefined otherwise.} \end{cases}$$

It is easy to verify that this satisfies (3. 1. 1). □

This gives us a notation for all finite trees as the following remark shows.

3. 1. 3 Remark

Let $F[\Sigma]$ denote the set of finite trees over Σ (trees with finite domain). Then $F[\Sigma]$ is the least set satisfying:

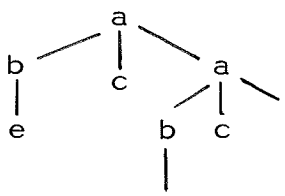
$\perp \in F[\Sigma]$

(3. 1. 2) If $\sigma \in \Sigma$ and $t_0,\ldots,t_{-1} \in F[\Sigma]$, then $\sigma(t_0,\ldots,t_{-1}) \in F[\Sigma]$.

Moreover, each $t \in F[\Sigma]$ can be written in the form of (3. 1. 2) in a unique way. □

This result allows us to define functions on $F[\Sigma]$ by induction on the complexity of the tree.

3. 1. 4 Example

The tree depicted $t =$  informally to the right is defined formally by:

$t[\epsilon] = a$ $\quad t[01] = e$

$t[0] = b$ $\quad t[21] = b$

$t[1] = c$ $\quad t[22] = c$

$t[2] = a$ $\quad t[\alpha] = \perp$ otherwise

and may be denoted by $t = a(b(e), c, a(b(\perp), c, \perp))$.

3.1.5 Theorem

(i) The relation $\leq$ on $T[\Sigma]$ defined by

(3.1.3) $t_1 \leq t_2$ iff for all $\alpha \in N$, $t_1[\alpha] = \perp$ or $t_1[\alpha] = t_2[\alpha]$

is a partial order (reflexive, transitive, and anti-symmetric).

(ii) $\perp \leq t$ for all $t \in T$

(iii) If $t_0 \leq t_1 \leq \ldots$ is a countable sequence of trees in $T[\Sigma]$, then there is a unique tree, denoted $\text{lub}\{t_i\}$ such that $t_i \leq \text{lub}\{t_i\}$ for all i and if $t_i \leq t$ for all i, then $\text{lub}\{t_i\} \leq t$.

Proof

(i) and (ii) are trivial. For (iii), $\text{lub}\{t_i\} = t$ where $t[\alpha] = \sigma$ if there is some $\sigma \neq \perp$ and some i such that $t_i[\alpha] = \sigma$. (There is at most one such σ.) $t[\alpha] = \perp$ otherwise. □

3.1.6 Definition

A set, together with an element $\perp$ and a relation $\leq$ satisfying the conditions of theorem 3.1.5 is called a cpo (chain-complete poset).

3.1.7 Theorem

Let $p_n: T[\Sigma] \to F[\Sigma]$ by $p_n(t)[\alpha] = \begin{cases} t[\alpha] & \text{if length } (\alpha) < n \\ \perp & \text{otherwise.} \end{cases}$

Given a tree $t \in T[\Sigma]$, let $\Psi(t)$ be the sequence $\langle t_0, t_1, \ldots \rangle$ of members of $F[\Sigma]$ defined by $t_i = p_i(t)$. Then

(i) $t_i \leq t_{i+1}$ for all i

(ii) $t_i = p_i(t_j)$ for all $j \geq i$

(iii) $t = \text{lub}\{t_i\}$

(iv) Ψ is an order isomorphism between $T[\Sigma]$ and the set of sequences over $F[\Sigma]$ satisfying (i) and (ii) and ordered componentwise. □

3. 1. 8 Remark

By the above theorem, we could take 3. 1. 3 as the definition of $F[\Sigma]$ and define $T[\Sigma]$ to be the set of sequences satisfying (i) and (ii) of 3. 1. 7. This is the approach taken in [15] and [16]. We prefer the method here (which was inspired by [5]) since it seems more intuitively accessible. □

3. 1. 9 Definition

If A and B are cpo's then $f:A \to B$ is continuous if $f(\bot) = \bot$, $x \leq y$ implies $f(x) \leq f(y)$, and $x_0 \leq x_1 \leq \ldots$ implies $\text{lub}\{t(x_i)\} = f(\text{lub}\{x_i\})$. $[A \to B]$ denotes the set of continuous functions from A to B, ordered point-wise, that is $f \leq g$ iff $f(x) \leq g(x)$ for all $x \in A$. $A \times B$ denotes the cortesian product of A and B ordered by $\langle x_1, y_1\rangle \leq \langle x_2, y_2\rangle$ iff $x_1 \leq x_2$ and $y_1 \leq y_2$. □

3. 1. 10 Proof

(i) $[A \to B]$ and $A \times B$ are cpo's.

(ii) If $f \in [A \to B]$ and $g \in [B \to C]$ then $g \circ f \in [A \to C]$.

(iii) If $f \in [A \to B]$ and $g \in [A \to C]$ then $f \times g \in [A \to B \times C]$ where $(f \times g)(x) = \langle f(x), g(x)\rangle$.

(iv) $\pi_j \in [A_0 \times \ldots \times A_{-1} \to A_j]$ where $\pi_j(x_0, \ldots, x_{-1}) = x_j$.

(v) If A is any set, then $\mathcal{P}(A) = \{B \mid B \subseteq A\}$ is a cpo, where $\bot = \emptyset$, $B \leq C$ iff $B \subseteq C$, and $\text{lub}\{B_i\} = \bigcup B_i$. If $f:A \to \mathcal{P}(B)$ is any function, then $\hat{f} \in [\mathcal{P}(A) \to \mathcal{P}(B)]$ where $\hat{f}(C) = \bigcup\{f(x) \mid x \in C\}$. If A is countable, then every $g \in [\mathcal{P}(A) \to \mathcal{P}(B)]$ is $\hat{f}$ for some $f:A \to \mathcal{P}(B)$. □

3. 1. 11 Remarks

(1) If S is any set, then $\hat{S} = S \cup \{\bot\}$ is a cpo, there $\bot$ is a new symbol and $x \leq y$ iff $x = y$ or $x = \bot$.

(2) $T[\Sigma]$ is a sub-cpo of $[\hat{\mathbb{N}}^* \to \hat{\Sigma}]$.

(3) If A and B are alphabets, then any substitution (as defined in [7, p. 124] or [1, p. 146]) $h: \mathcal{P}(A^*) \to \mathcal{P}(B^*)$ is continuous by 3. 1. 1(v). □

3. 2 Substitution Operators

We stated above that under the assignment of functions to modal variables, each expression e_i denotes a function "in an obvious way". We now make this more precise.

Recall from 3. 1. 2 that each $\sigma \in \Omega$ also denotes a function $\sigma: T^{\text{rank}(\sigma)} \to T$.

3. 2. 1 Lemma

σ is continuous. □

3. 2. 2 Definition

Let $e = \langle \bar{e}_0, \ldots, e_{-1} \rangle$ be a sequence of expressions – i. e. members of $F[\Omega \cup V \cup X]$ and let $\bar{f} = \langle f_0, \ldots, f_{-1} \rangle$ be a sequence of functions where $f_i: T^{\text{rank}(v_i)} \to T$. Then $\bar{e} \leftarrow \bar{f}$ is the function defined by induction on the complexity of $\bar{e}$ as follows:

$$\bar{e} \leftarrow \bar{f} = (e_0 \leftarrow \bar{f}) \times \ldots \times (e_{-1} \leftarrow \bar{f})$$

$$\sigma(e_0, \ldots, e_{-1}) \leftarrow \bar{f} = \sigma \cdot (\bar{e} \leftarrow \bar{f})$$

$$v_i(e_0, \ldots, e_{-1}) \leftarrow \bar{f} = f_i \circ (\bar{e} \leftarrow \bar{f})$$

$$(x_i \leftarrow f) = \pi_i$$

(By 3. 1. 3 this defines $\bar{e} \leftarrow \bar{f}$ completely.) □

The reader may find this definition easier to understand if he notices that $(f \cdot (g_1 \times g_2 \times \ldots \times g_{-1}))(x) = f(g_0(x), \ldots, g_{-1}(x))$.

3.2.3 Lemma

(i) If each f_i is continuous then $\bar{e} \leftarrow \bar{f}$ is continuous.

(ii) The operator $\mathcal{F}$ defined by $\mathcal{F}(\bar{f}) = \bar{e} \leftarrow \bar{f}$ is continuous – i.e. $\mathcal{F} \in [D_0 \times D_1 \times \ldots \times D_{-1} \to D_0 \times D_1 \times \ldots \times D_{-1}]$ where $D_i = [T^{rank(v_i)} \to T]$.

Proof

(i) follows from 3.1.10(ii, iii, and iv) and 3.2.1.

(ii) follows from the fact that the operators $\cdot$ and $\times$ are continuous. □

3.2.4 Theorem (Tarski [17])

If A is any cpo and $f \in [A \to A]$, then f has a fixed point, an element $x \in A$ such that $f(x) = x$. In fact, f has at least fixed point, denoted Yf, which may be computed by

$$f_0 = \underline{\perp}$$
$$f_{n+1} = f(f_n)$$
$$Yf = \text{lub}\{f_n\}$$

3.2.5 Corollary

For each definition $\underline{\underline{D}}$ as in (2.1.1), there is a sequence $F = \langle f_0, \ldots, f_{-1} \rangle$ such that $\bar{f} = \bar{e} \leftarrow \bar{f}$. □

This is the promised semantics for type definitions.

Summarizing, a mode definition $\underline{\underline{D}}$ is of the form $\{v_i(x_0, \ldots, x_{-1}) = e_i \mid i = 0, 1, 2, \ldots\}$. A solution to the sequence of equations is

a sequence $\bar{f}$ of continuous functions such that $f_i(t_0, \ldots, t_{-1}) =$ $= (e_i \leftarrow \bar{f})(t_0, \ldots, t_{-1})$ for all i and all $t_0, \ldots, t_{-1}$.

3.2.6 Notation

We will say that the solution of $\underline{\underline{D}}$ is the least fixed point Y computed as in theorem 3.2.4, and the function defined by $\underline{\underline{D}}$ is the first component of this fixed point, $\pi_0(Y\ \mathcal{F})$. Finally, and most important, two definitions are equivalent if they define the same function.

3.3 Functions and Trees

In 3.1.2 we used a symbol $\sigma \in \Sigma$ to denote a function. More generally, many (not all) functions on $T[\Sigma]$ can be represented by trees.

3.3.1 Definition

Let $X_n = \{x_0, \ldots, x_{n-1}\}$ and let $t \in T[\Omega \cup X_n]$. Then for all $m \geq n$ and all $\Sigma \supseteq \Omega$, t denotes a function fun(t): $T[\Sigma]^m \rightarrow T[\Sigma]$ where fun(t) $(t_0, \ldots, t_{m-1})$ is the result of replacing each occurrence of x_i by t in t_i. More precisely, let s be the tree:

$$s[\alpha] = t[\alpha] \text{ if } t[\alpha] \in \Omega$$

$$s[\alpha\beta] = t_i[\beta] \text{ if } t[\alpha] = x_i$$

It is straightforward to check that this defines a unique tree. Let fun(t) $(t_0, \ldots, t_{m-1}) = s$. □

Notice that this definition extends 3.1.2 in the sense that $\sigma = \text{fun}(\sigma(x_0, \ldots, x_{-1}))$ (as a function). Next, we extend this notation of "replacing occurrences of the symbol τ in t with the tree s" to the case in which rank(τ) $\neq 0$ - i.e. in which τ can appear at an interior node.

3.3.2 Definition

Let $\bar{e} = \langle e_0, \ldots, e_{-1} \rangle$ and $\bar{t} = \langle t_0, \ldots, t_{-1} \rangle$ be finite sequences of trees where $e_i \in F[\Omega \cup V \cup X]$ and $t_i \in T[\Omega \cup V \cup X]$. Then $\bar{e} \leftarrow \bar{t}$, the result of replacing v_i by t_i in $\bar{e}$ is defined by induction on the complexity of $\bar{e}$ as follows:

$$\bar{e} \leftarrow \bar{t} = \langle e_0 \leftarrow \bar{t}, \ldots, e_{-1} \leftarrow \bar{t} \rangle$$

$$\sigma(e_0, \ldots, e_{-1}) \leftarrow \bar{t} = \sigma(\bar{e} \leftarrow \bar{t})$$

$$v_i(e_0, \ldots, e_{-1}) \leftarrow \bar{t} = \mathrm{fun}(t_i)\,(\bar{e} \leftarrow \bar{t})$$

$$x_i \leftarrow \bar{t} = x_i$$

(See, for example figure 1b or 2a.) □

3.3.3 Remark

The restriction that each e_i be finite is not essential. We could define $\bar{e} \leftarrow \bar{t}$ directly (rather than inductively) for arbitrary $\bar{e}$, but the definition would be much more complicated, and we are only interested in the case of finite e_i. □

In view of the similarity between definitions 3.2.2 and 3.3.2 it is not surprising that we have the following result.

3.3.4 Lemma

$\mathrm{fun}(\bar{e} \leftarrow \bar{t}) = \bar{e} \leftarrow \mathrm{fun}(\bar{t})$. □

3.3.5 Remark

In the statement of 3.3.4 we implicitly extended the domain of fun from trees to tuples of trees by letting $\mathrm{fun}(\langle t_0, \ldots, t_{-1} \rangle) = \langle \mathrm{fun}(t_0), \ldots, \mathrm{fun}(t_{-1}) \rangle$. We will continue to make such extensions without explicit mention. □

3.3.6 Lemma

(i) fun: $T[\Omega \cup X_n] \to [T^n \to T]$ is continuous

(ii) fun is one-to-one

(iii) $\leftarrow$ as an operation on $T[\Omega \cup V \cup X]$ is associative wherever it is defined. That is, $\bar{e} \leftarrow (\bar{e}' \leftarrow \bar{t}) = (\bar{e} \leftarrow \bar{e}') \leftarrow \bar{t}$.

3.3.7 Theorem

The minimal fixed point $Y\mathcal{F}$ of 3.2.3 can be described by trees. More precisely, $Y\mathcal{F} = \text{fun}(\bar{t})$ where $\bar{t} = \text{lub}\{\bar{t}^i\}$ and $\bar{t}^i$ is defined by

$\bar{t}^0 = \langle \underline{\perp}, \ldots, \underline{\perp} \rangle$

$\overline{t^{n+1}} = \bar{e} \leftarrow \bar{t}^n$

Proof

By 3.2.4, $Y\mathcal{F} = \text{lub}(\mathcal{F}^n(\underline{\perp}))$ where $\underline{\perp}$ is the nowhere defined function.

We will show by induction on n that $\mathcal{F}^n(\underline{\perp}) = \text{fun}(\overline{t^n})$:

$\mathcal{F}^0(\underline{\perp}) = \underline{\perp}$ (where $\underline{\perp}$ is the nowhere defined function)

$= \text{fun}(\bar{\perp})$ (where $\perp$ is the nowhere defined tree)

$= \text{fun}(\overline{t^0})$

$\mathcal{F}^{n+1}(\underline{\perp}) = \mathcal{F}(\mathcal{F}^n(\underline{\perp}))$

$= \bar{e} \leftarrow \mathcal{F}^n(\underline{\perp})$ by definition of $\mathcal{F}$

$= \bar{e} \leftarrow \text{fun}(\overline{t^n})$ by induction hypothesis

$= \text{fun}(\bar{e} \leftarrow \overline{t^n})$ by lemma 3.3.4

$= \text{fun}(\overline{t^{n+1}})$ by definition of $\overline{t^n}$

Hence $Y\mathcal{F} = \text{lub}(\mathcal{F}^n(\underline{\perp}))$

$= \text{lub}\{\text{fun}(\overline{t^n})\}$

$= \text{fun}(\text{lub}\{\overline{t^n}\})$ by 3.3.6 (i)

$= \text{fun}(\bar{t})$ □

3.3.8 Corollary

The function defined by definition $\underline{\underline{D}}$ is $fun(\pi_0(lub\{\overline{t^n}\}))$. □

3.3.9 Notation

Let $\underline{\underline{D}}$ be a definition, and $\bar{t} = lub\{\overline{t^n}\}$ be derived from $\underline{\underline{D}}$ as in 3.3.7. Then $\bar{t}$ is said to be the tuple of trees defined by $\underline{\underline{D}}$. The tree defined by $\underline{\underline{D}}$ is the first component of this tuple, $\pi_0(\bar{t})$. □

3.3.10 Remark

Although a definition $\underline{\underline{D}}$ technically defines a function, by virtue of 3.3.8 we can pretend that $\underline{\underline{D}}$ defines a tree: If f_0 is the function defined by $\underline{\underline{D}}$ and t_0 is the tree defined by $\underline{\underline{D}}$ then $f_0 = fun(t_0)$. Since fun is one-to-one, two definitions are equivalent iff they define the same tree. □

3.3.11 Example

Return to example 1.2. The meaning of figure 1b can now be made more precise. Notice that the tuples $\overline{t^n}$ of 3.3.7 have the property that $\overline{t^n} = \overline{e^n} \leftarrow \langle \underline{1}, \ldots, \underline{1} \rangle$ where $\overline{e^n} = \bar{e} \leftarrow \bar{e} \leftarrow \ldots \leftarrow \bar{e}$ (n factors) or (by induction)

$$\overline{e^0} = \langle v_0(x_0, \ldots, x_{-1}), \ldots, v_{-1}(x_0, \ldots, x_{-1}) \rangle$$

$$\overline{e^{n+1}} = \bar{e} \leftarrow \overline{e^n} = \overline{e^n} \leftarrow \bar{e} \text{ (by the associativity of } \leftarrow).$$

The first 4 trees of figure 1b represent e_0^0, $e_0^1 = e_0$, $e_0^2 = e_0 \leftarrow \bar{e}$, and $e_0^3 = e_0 \leftarrow \bar{e} \leftarrow \bar{e}$. (Since v_1 does not appear in e_0, we need not show e_1.) The last tree in 1b represents t_0, and the tree of 1c represents $fun(t_0)$ (int) $= t_1$. □

4 TREES AND LANGUAGES

In this section, we establish an intimate connection between modals and deterministic context-free languages, and use this connection to prove the main results of the paper
The first step has already been taken by noting (in 3.3.10) that we can confine our attention to trees. The next step is to notice that the trees involved all have finite range.

4.1 Lemma

All trees mentioned in theorem 3.3.7 have finite range. (The range of t is $\{\sigma \mid t[\alpha] = \sigma \text{ for some } \alpha \in \mathbb{N}^*\}$).

Proof

Clearly, if t is any of these trees, then $t[\alpha] = \sigma$ only if σ appears somewhere in $\underline{\underline{D}}$. But $\underline{\underline{D}}$ contains a finite number of symbols.

□

Now let Σ be a (not necessarily finite) ranked set and let $t \in T[\Sigma]$. Suppose the range of t is finite. Let $n = \max\{\text{rank}(\sigma) \mid \sigma \in \text{range}(t)\}$, and let $[n] = \{0, 1, \ldots, n-1\}$. For each $\sigma \in \text{range}(t)$, $t^{-1}[\sigma] \subseteq [n]^*$; that is, $t^{-1}[\sigma]$ is a language over the finite alphabet $[n]$. $t^{-1}[\sigma]$ may be thought of as "the set of addresses of nodes where σ lives". This finite collection of languages completely characterizes t. This observation is so important in what follows, that we state it as a theorem.

4.2 Theorem

If $t \in T[\Sigma]$ has a finite range, then there is a finite alphabet $[n] \subseteq \mathbb{N}$ and a finite collection of disjoint languages $L_\sigma \subseteq [n]^*$ such that that

(4.1) $t[\alpha] = \sigma$ iff $\alpha \in L_\sigma$

and $t[\alpha] = \perp$ iff $\alpha \notin \bigcup\{L_\sigma\}$. □

In [16] we showed that for ordinary type definitions (i.e. those without parameters), these languages are all regular sets, and, conversely, that given any finite collection of regular sets such that (4.1) defines a tree t, t is definable by some definition. Notice, however, that the definable tree t shown in 1c has the property that $t^{-1}[\underline{\text{int}}] = \{ 1^n 0^{n+1} \mid n \geq 0 \}$ which is not a regular set [7].

4.1 Description Languages

In the previous section we showed how a tree can be described by a finite collection of languages. Here we introduce a device for describing a tree by a single language.

4.1.1 Definition

Let Σ be a ranked set and let $\hat{\Sigma} = \Sigma \cup \{\langle \sigma, n \rangle \mid \sigma \in \Sigma \text{ and } n < \text{rank}(\sigma)\}$.

Let $\mathcal{D}: T[\Sigma] \to \mathcal{P}(\hat{\Sigma}^*)$ by

$$\mathcal{D}(t) = \{\langle \sigma_0, n_0 \rangle \ldots \langle \sigma_{k-1}, n_{k-1} \rangle \sigma_k \mid k \geq 0 \text{ and } t[n_0, \ldots, n_{j-1}] = \sigma_j \text{ for all } j \leq k\}.$$

Note that if Σ is infinite, then $\hat{\Sigma}$ is infinite, but if range(t) is finite, then $\mathcal{D}(t) \subseteq A^*$ for some finite subset $A \subseteq \hat{\Sigma}$. $\mathcal{D}(t)$ is called the description language of t.

4. 1. 2 Lemma

$\mathfrak{D}$ is continuous. □

4. 1. 3 Definition

For each $\sigma \in \Sigma$, let σ be the finite substitution ([7, p. 124] or [1, p. 196]) defined by

$$h_\sigma(<\tau, i>) = \{i\} \text{ for all } \tau \in \Sigma$$

$$h_\sigma(\tau) = \emptyset \text{ if } \tau \neq \sigma$$

$$h_\sigma(\sigma) = \{\epsilon\}.$$

Since $h_\sigma(\omega)$ contains at most one string for any string ω, we will regard h_σ as a partial function and write $h_\sigma(\omega) = \alpha$ rather than $h_\sigma(\omega) = \{\alpha\}$. □

4. 1. 4 Lemma

$$t^{-1}[\sigma] = h_\sigma(\mathfrak{D}(t)).$$ □

4. 1. 5 Corollary

$$t_1 = t_2 \text{ iff } \mathfrak{D}(t_1) = \mathfrak{D}(t_2).$$ □

We have already given two meanings to the operator $\leftarrow$. Now we give a third.

4. 1. 6 Definition

Let $\hat{\Sigma}$ be as in 4. 1. 1 where $\Sigma = \Omega \cup V \cup X$. Let $M_0, \ldots, M_{-1} \subseteq \hat{\Sigma}^*$, where the number of M's is $||V||$ [+]. Let Ψ be the substitution defined by

---

[+] For any set S, $||S||$ is the cardinality of S.

$$\Psi(v_j) = M_j - \hat{\Sigma}^* X$$
$$= \{\alpha \in M_j \mid \alpha \neq \beta x_i \text{ for any } \beta \text{ and } i\}$$
$$\Psi(\langle v_j, i\rangle) = M_j/\{x_i\} = \{\alpha \in \hat{\Sigma}^* \mid \alpha x_i \in M_j\}$$
$$\Psi(\tau) = \{\tau\} \text{ for any other } \tau \in \hat{\Sigma}.$$

Let $L_0, \ldots, L_{-1} \subseteq \hat{\Sigma}^*$. Then $\langle L_0, \ldots, L_{-1}\rangle \leftarrow \langle M_0, \ldots, M_{-1}\rangle =$
$= \langle \Psi(L_0), \ldots, \Psi(L_{-1})\rangle$. □

The reason for "overloading" the symbol ← is the following:

4.1.7 Proposition

Let $\bar{t}$ and $\bar{e}$ be tuples of trees in $T[\Omega \cup V \cup X]$. Then
$\mathcal{D}(\bar{t} \leftarrow \bar{e}) = \mathcal{D}(\bar{t}) \leftarrow \mathcal{D}(\bar{e})$. □

4.1.8 Corollary

Let t be the tree defined by the definition $\underline{\underline{D}}$. Then $\mathcal{D}(t) = \pi_0(\bar{L})$
where

$$\overline{L^0} = \langle \emptyset, \ldots, \emptyset\rangle$$
$$\overline{L^{n+1}} = \mathcal{D}(\bar{e}) \leftarrow \overline{L^n}$$
and $$\bar{L} = \bigcup_{n=0}^{\infty} \overline{L^n}.$$

Proof

By 3.3.7, 4.1.2, and the fact that $\mathcal{D}(\underline{1}) = \emptyset$. □

4.2 From Definitions to Pushdown Automata.

In this section we show how, given a definition $\underline{\underline{D}}$ and a symbol σ appearing in $\underline{\underline{D}}$ we can construct a deterministic pushdown automaton (DPDA) P such that $L(P) = t_0^{-1}[\sigma]$. Our notation for DPDA's is exactly as in [1] (q.v) except that we use ¢ rather than e for the null string. The description languages introduced in section 4.1

are not used directly in the construction, but are used in the proof of correctness of the construction. First we present the construction, then we give an example. [A detailed correctness proof is omitted in this draft.]

4.2.1 Construction

Let $\underline{\underline{D}}$ be a definition as in (2.1.1) and let σ be a symbol appearing in $\underline{\underline{D}}$. Let $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ where

Q (the set of states) $= \{\langle i,\alpha\rangle \mid \alpha \in \mathrm{dom}(e_i)\}$

(A state is a node in some tree of $\underline{\underline{D}}$.)

Σ (the input alphabet) $= \{0,\ldots,m-1\}$ where

$m = \max\{\mathrm{rank}(\tau) \mid \tau \in \Omega$ appears in $\underline{\underline{D}}\}$

Γ (the stack alphabet) $= \{\langle i,\alpha\rangle \in Q \mid e_i[\alpha] \in V\} \cup Z_0$

(A stack symbol is a node labeled by a modal variable.)

q_0 (the initial state) $= \langle p, \epsilon\rangle$ (the root of the first tree in $\underline{\underline{D}}$)

F (the set of final states) $= \{\langle j,\alpha\rangle \in Q \mid e_j[\alpha] = \sigma\}$

(A final state is a node labeled σ.)

and δ (the transition function) is defined as follows:

(i) If $e[i] \in \Omega$, then $\delta(\langle i,\alpha\rangle, j, Z) = (\langle i,\alpha j\rangle, Z)$ $\forall Z \in \Gamma$

(ii) If $e[i] = v_k$, then $\delta(i,\alpha\rangle,\epsilon, Z) = (\langle k,\epsilon\rangle,\langle i,\alpha\rangle, Z)$ $\forall Z \in \Gamma$

(iii) If $e[i] = x_k$, then $\delta(\langle i,\alpha\rangle,\epsilon,\langle j,\beta\rangle) = (\langle j,\beta k\rangle,\epsilon)$

4.2.2 Example

Let σ, τ, $\rho \in \Omega$ and let $\mathrm{rank}(\sigma) = 3$, $\mathrm{rank}(\tau) = \mathrm{rank}(\rho) = 0$. Let $\underline{\underline{D}}$ be the definition $v_0(x_0,x_1) = \sigma(v_0(\sigma(\tau,x_0,\rho),\sigma(\tau,\rho,x_1)),x_0,x_1)$. Let t be the tree defined by $\underline{\underline{D}}$. The derivation of t is illustrated in figure 2a. Figures 2b and 2c illustrate the action of P on input 00110. $L(P) = t^{-1}[\tau] = \{0^m1^n0 \mid m \geq n \geq 1\} \cup \{0^m2^n0 \mid m \geq n \geq 1\}$. (Notice that L(P) is not an LL language [1]).

4.2.3 Theorem

Let P be the DPDA constructed in 4.2.1. Then $L(P) = t^{-1}[\sigma]$ where t is the tree defined by $\underline{\underline{D}}$. □

4.2.4 Corollary

If the DPDA equivalence problem is decidable, then the type equivalence problem is decidable.

Proof

Given two definitions $\underline{\underline{D}}$ and $\underline{\underline{D}}'$ $\underline{\underline{D}}$ is equivalent to $\underline{\underline{D}}'$ iff $t_0 = t_0'$ where t_0 and t_0' are the trees defined by $\underline{\underline{D}}$ and $\underline{\underline{D}}'$, by 3.3.10. For each symbol σ appearing in $\underline{\underline{D}}$ or $\underline{\underline{D}}'$, construct DPDA's P_0 and P_0' according to 4.2.1. Then $t_0 = t_0'$ iff $L(P_\sigma) = t_0^{-1}[\sigma] = t_0'^{-1}[\sigma] = L(P_\sigma')$ for each σ. □

4.3 From Pushdown Automata to Definitions

By theorem 4.2.3, if t is a definable tree, then there is a finite set of deterministic languages $\{L_\sigma\}$ which completely characterize t in the sense of (4.1). Unfortunately, the converse is not true. Even if t has finite range and $t^{-1}[\sigma]$ is deterministic context-free for all σ, it may not be the case that t is definable. Fortunately, the converse of 4.2.4 does not need the full converse of 4.2.3; a weaker result suffices. Given any deterministic context-free language L (satisfying certain properties), there is a definable tree t built up from the symbols σ and τ such that $t^{-1}[\tau] = L$ and $t^{-1}[\sigma]$ is completely determined by L. The first step is to reduce an arbitrary DPDA to an easily manageable form. Fortunately, this has already been done in [1].

4.3.1 Definition (quoted from [1, p. 691]

A DPDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is in normal form if it has

all the following properties:

(1) P is loop-free. Thus, on each input, P can make only a bounded number of moves.

(2) F has a single member, q_f, and if $(q_0, w, Z_0) \vdash^* (q_f, \epsilon, \gamma)$, then $\gamma = Z_0$. That is, if P accepts an input string, then P is in the final state q_f and the pushdown list consists of the start symbol alone.

(3) Q can be written as $Q = Q_s \cup Q_w \cup Q_e \cup \{q_f\}$, where Q_s, Q_w, and Q_e are disjoint sets, called the <u>scan</u>, <u>write</u>, and <u>erase</u> states, respectively; q_f is in none of these three sets. The states have the following properties:

(a) If q is in Q_s, then for each $a \in \Sigma$, there is some state p_a such that $\delta(q, a, Z) = (p_a, Z)$ for all Z. Thus, if P is in a scan state, the next move is to scan the input symbol. In addition, this move is always independent of the symbol on top of the pushdown list.

(b) If q is in Q_w, then $\delta(q, \epsilon, Z) = (p, YZ)$ for some p and Y and for all Z. A write state always prints a new symbol on top of the pushdown list, and the move is independent of the current input symbol and the symbol on top of the pushdown list.

(c) If q is in Q_e, then for each $Z \in \Gamma$, there is some state p_Z such that $\delta(q, \epsilon, Z) = (p_Z, \epsilon)$. An erase state always removes the topmost symbol from the pushdown list without scanning a new input symbol.

(d) $\delta(q, a, Z) = \emptyset$ for all a in $\Sigma \cup \{\epsilon\}$ and $Z \in \Gamma$. No moves are possible in the final state.

(4) If $(q, w, Z) \vdash^+ (p, \epsilon, Z)$, then $w \neq \epsilon$. That is, a sequence of moves which (possibly) enlarges the stack and returns to the same level cannot occur on ϵ input. A sequence of moves $(q, w, Z) \vdash^+ (p, \epsilon, Z)$ will be called a <u>traverse</u>. Note that the possibility or impossibility of a

traverse for given q, p, and w is independent of Z, the symbol on top of the pushdown list,

In short, a scan state reads the next input symbol, a write state prints a new symbol on the stack, and an erase state examines the top stack symbol, erasing it. Only scan states may shift the input head. □

4.3.2 Theorem (quoted from [1, p. 691])

If $L \subseteq \Sigma^*$ is a deterministic language, and ¢ is not in Σ, then L¢ is L(P) for some DPDA P in normal form. □

We will assume henceforth that $\Sigma = \{0, 1, \ldots, n-1\}$ for some n.

4.3.3 Definition

Let P be a DPDA in normal form. Then the function δ may be characterized by three functions: the scan goto function f, the push-pop function g, and the push-goto function h defined as follows:

$f : Q_s \cup \Sigma \to Q$ where for all $Z \in \Gamma$, $q_i \in Q_s$, $n \in \Sigma$

$\delta(q_i, n, Z) = (f(q_i, n), Z)$

$g : Q_w \times Q_e \to Q$ and $h : Q_w \to Q - Q_e$ where for all $Z \in \Gamma$,

$q_i \in Q_w$, $p_j \in Q_e$

$\delta(q_i, \epsilon, Z) = (h(q_i), Y_i Z)$ for some $Y_i \in \Gamma$ depending only on q_i

and $\delta(p_j, \epsilon, Y_i) = (g(q_i, p_j), \epsilon)$

(Notice that $h(q_i) \notin Q_e$ since otherwise there would be a traverse

$(q_i, \epsilon, Z) \vdash (h(q_i), \epsilon, Y_i Z) \vdash (g(q_i, p_j), \epsilon, Z)$

contradicting point (4) of 4.3.1. □

Now we can describe the construction of a type definition from a normal form DPDA P.

4.3.4 Construction

Let P be a DPDA in normal form. Without loss of generality,

we may assume that $q_0 \in Q_e$. Let $\Omega = \{ \sigma, \tau \}$ where rank(σ) = $||\Sigma||$ and rank(τ) = 0. Let $V = \{ v_0, \ldots, v_{-1} \}$ where $||V|| = ||Q - Q_e||$ and rank(v_i) = $||Q_e||$ for each i. Let $\bar{x}$ denote $\langle x_0, \ldots, x_{||Q_e||-1} \rangle$. Arbitrarily order Q_e and $Q - Q_e$ so that $Q_e = \{ p_0, \ldots, p_{-1} \}$, $Q - Q_e = \{ q_0, \ldots, q_{-1} \}$ and q_0 is the initial state.

For each q_i, let $k(q_i)$ be the expression (tree) "$v_i(\bar{x})$" $\in T[\Omega \cup V \cup X]$ and for each p_i, let $k(p_i)$ be the expression "x_i" $\in T[\Omega \cup V \cup X]$.

Let $\underline{\underline{D}}$ be the definition

$$v_0(\bar{x}) = e_0$$
$$\vdots$$
$$v_{-1}(\bar{x}) = e_{-1}$$

where

- if $q_i \in Q_s$ then $e_i = \sigma(k(f(q_i, 0)), \ldots, k(f(q_i, -1)))$
- if $q_i \in Q_w$ then $e_i = v_j(k(g(q_i, p_0)), \ldots, k(g(q_i, p_{-1})))$
  where $q_j = h(g_i)$
- if $q_i = q_f$ then $e_i = \tau$. □

4.3.5 Example

Let P be the normal form DPDA in figure 3a. L(P) = $= \{ 0^n 1 0^n 1 \mid n \geq 0 \}$. Then the functions used and the definition constructed according to 4.3.4 are illustrated in figure 3b. □

4.3.6 Theorem

Let P be a DPDA in normal form and let $\underline{\underline{D}}$ be the definition constructed from P by 4.3.4. Let t_0 be the tree defined by $\underline{\underline{D}}$. Then

$$t_0[\alpha] = \begin{cases} \tau & \text{if } \alpha \in L(P) \\ \perp & \text{if } \alpha \notin L(P) \text{ but } \alpha = \beta\gamma \text{ for some } \beta \in L(P) \\ \sigma & \text{otherwise} \end{cases}$$ □

4.3.7 Corollary

If the type equivalence problem is decidable, then the DPDA equivalence problem is decidable.

Proof

Let $L, L' \subseteq \Sigma^*$ be deterministic languages where $\Sigma = \{0, \ldots, n-1\}$. By 4.3.2, we can construct DPDA's P and P' such that $L(P) = Ln \subseteq (\Sigma \cup \{n\})^*$ and $L(P') = L'n \subseteq (\Sigma \cup \{n\})^*$. Let $\underline{\underline{D}}$ and $\underline{\underline{D}}'$ be definitions defining trees t_0 and t_0', constructed according to 4.3.4. Then

$\underline{\underline{D}}$ is equivalent to $\underline{\underline{D}}'$

iff $t_0 = t_0'$ (by 3.3.10)

iff $L(P) = L(P')$ (by 4.3.6)

iff $L = L'$ □

5. SUMMARY AND CONCLUSIONS

We have presented a modest extension to the usual type definition facilities found in current programming languages. We have given a precise mathematical semantics to this extended facility using a so-called "lattice theoretic" or "Scott-like" model. Using this model, we were able to show an intimate connection between definable types and deterministic context-free languages. In particular, we have shown that the equivalence problem for definable types is of equal difficulty with the equivalence problem for deterministic languages. Since the latter problem remains open in the face of several

years of concentrated efforts of researchers to close it, we must either abandon attempts to decide equivalence of types or restrict the range of types that may be defined.

What are the practical implications of this result? As we mentioned in § 1.1, we can avoid the problem entirely by refusing to consider separately declared types equivalent, or to consider them equivalent only if the declarations are of "essentially the same form" in some sense. On the other hand, it may be that even the modest extension we presented here is unnecessarily strong. For example, definitions (1.1), (1.2), and (1.3), which were presented as motivation for modals, define regular types and thus could be defined without the use of parameters. In this case, the parameters serve merely as a convenience, allowing various aspects of the type intlist to be specified separately. This one could require that all types defined be regular (in the sense that $t^{-1}[\sigma]$ is a regular set for all σ). In view of the fact that it is decideable whether a given deterministic context-free set is regular [7, p. 230], it should not be hard to show that there is an algorithm to enforce this restriction. It remains to be seen whether non-regular types are of any practical use.

In fact, a somewhat stronger restriction, which is sufficient to ensure that all types declared are regular, is to change the syntax so that actual parameters to modal variables must be parameter symbols or modal constants of rank 0. This still allows definitions such as (1.1) but would be easier to enforce than the restriction of the previous paragraph. More experience with modals is necessary before we can state whether such restrictions prohibit any truly useful type definitions.

Acknowledgements

The results reported in this paper were obtained with the invalueable help and encouragement of Alan Demers. The construction of 4.3.4 was inspired by the proof of theorem 4.2.4 in [3]. We feel that the content of § 4 is essentially the same as that of [2], although the parallel is not exact, and we obtained our results before we were aware of [2]. Portions of this research were completed while the author was a student at Cornell University, Ithaca, N.Y.

References

[1] Aho, A.V. and Ullman, J.D. The Theory of Parsing, Translation and Compiling. Prentice-Hall, Englewood Cliffs, N.J., 1973.

[2] Courcell,B. Recursive schemes, algebraic trees, and deterministic languages. IEEE Symp. on Switching and Automata Theory, Vol. 15, 1974, pp. 52-62.

[3] Fischer, M.J. Grammars with Macro-like Productions. Ph.D. Thesis. Harvard University, 1968.

[4] Gehani, N. Data Types for Very High Level Languages (Ph.D. Thesis). Tech. Rep. TR75-258, Computer Sci. Dept., Cornell U., Ithaca, N.Y., 1975.

[5] Goguen, J.A. and Thatcher, J.W. Initial algebra semantics. IEEE Symp. on Swithching and Automata Theory, Vol. 15, 1974, pp. 63-77.

[6] Gries, D. and Gehani, N. Some ideas on data types in high level languages. Tech. Rep. TR75-244, Computer Sci. Dept., Cornell U., Ithaca, N.Y., 1975.

[7] Hopcroft, J.E. and Ullman, J.D. Formal Languages and their Relation to Automata. Addison-Wesley, Reading, Mass., 1969.

[8] IFIP Working Group 2.1. Report of the subcommittee on data-processing and transport-modals, with application to sorting. Algol Bulletin AB37.4.3 (May 1971).

[9] Jensen, K. and Wirth, N. Pascal User Manual and Report. Lecture Notes in Computer Science. Vol. 18. Springer Verlag, Berlin, 1974.

[10] Kral, J. The equivalence of modes and the equivalence of finite automata. Algol Bulletin AB35.4.5 (March 1973).

[11] Lewis, C.H. and Rosen, B.K. Recursively defined datatypes, part I. ACM Symposium on Principles of Programming Languages, 1973.

[12] Lindsey, C.H. Modals. Algol Bulletin AB37.4.3 (1974).

[13] Rosen, B.K. Tree-manipulating systems and Church-Rosser theorems. J. ACM, 20, 1 (January, 1973), 160-187.

[14] Rosen, B.K. and Lewis, C.H. Recursively defined datatypes, part II. Report RC4713, IBM J.J. Watson Research Center, Yorktown Hts., N.Y., 1974.

[15] Scott, D. The Lattice of flow diagrams, in Semantics of Algorithmic Languages, E. Engler, ed., Lecture Notes in Mathematics, Vol. 188, Springer Verlag, Berlin, 1971.

[16] Solomon, M. Modes, values and expressions. ACM Symp. on Principles of Programming Languages, Vol. 2, 1975, pp. 149-159. (Also available as Tech. Rep. TR74-219, Computer Sci. Dept., Cornell U., Ithaca, N.Y., 1974).

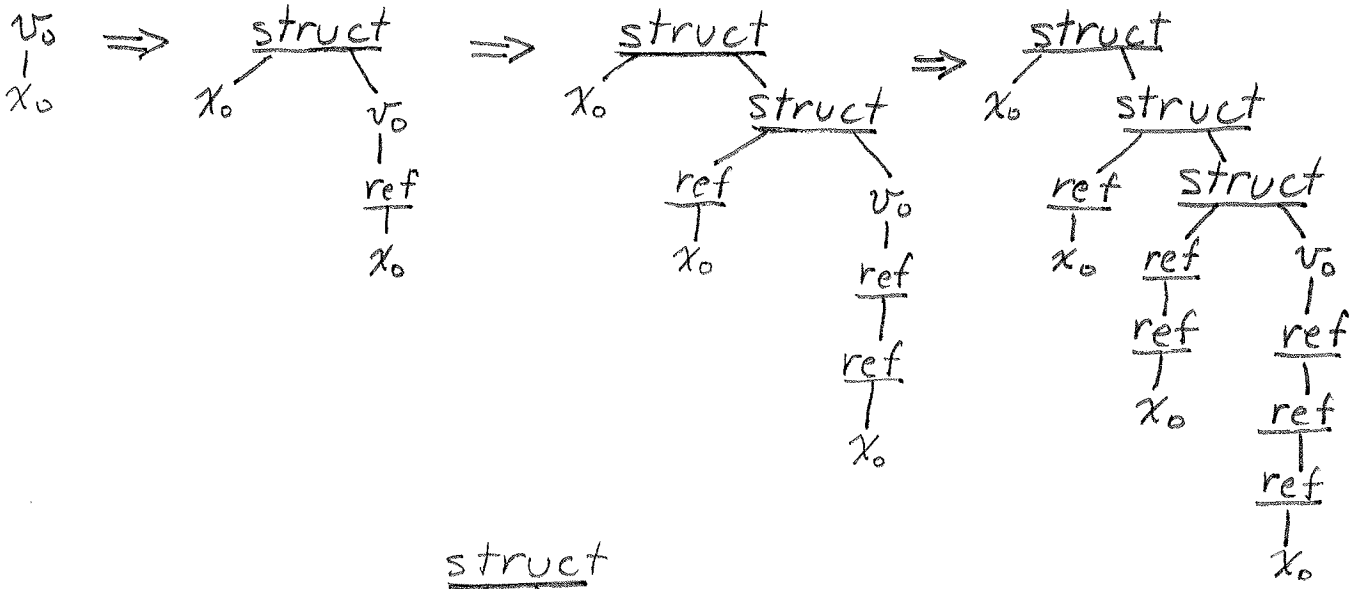
[17] Tarski, A. A lattice-theoretic fixpoint theorem and its applications. Pacific I. of Math. 5 (1955) 285-309.

[18] Wijngaarden, A. von et al. Revised report on the algorithmic language ALGOL 68. Tech. Rep. TR74-3, Computing Sci. Dept., U. of Alberta, Edmonton, Alberta, 1974.

$$v_0(\chi_0) = \underline{\text{struct}}\ (\chi_0, v_0(\underline{\text{ref}}\ (\chi_0)))$$

$$v_1 = v_0\ (\underline{\text{int}})$$

Figure 1a



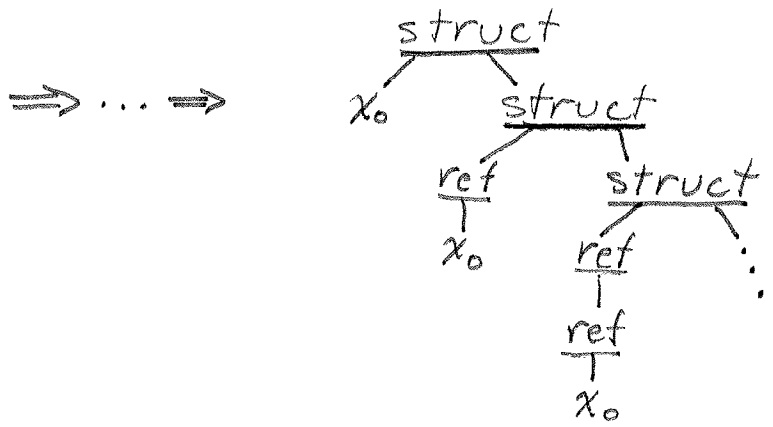


Figure 1b

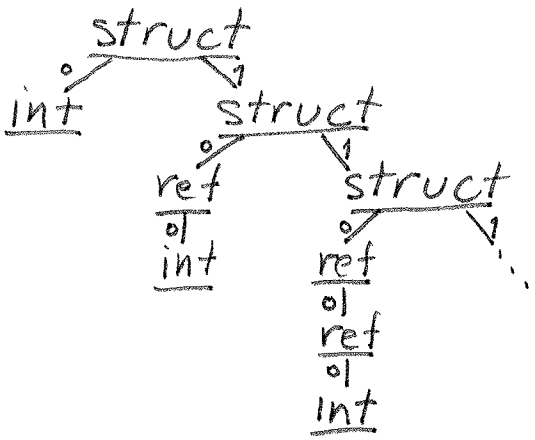


Figure 1c

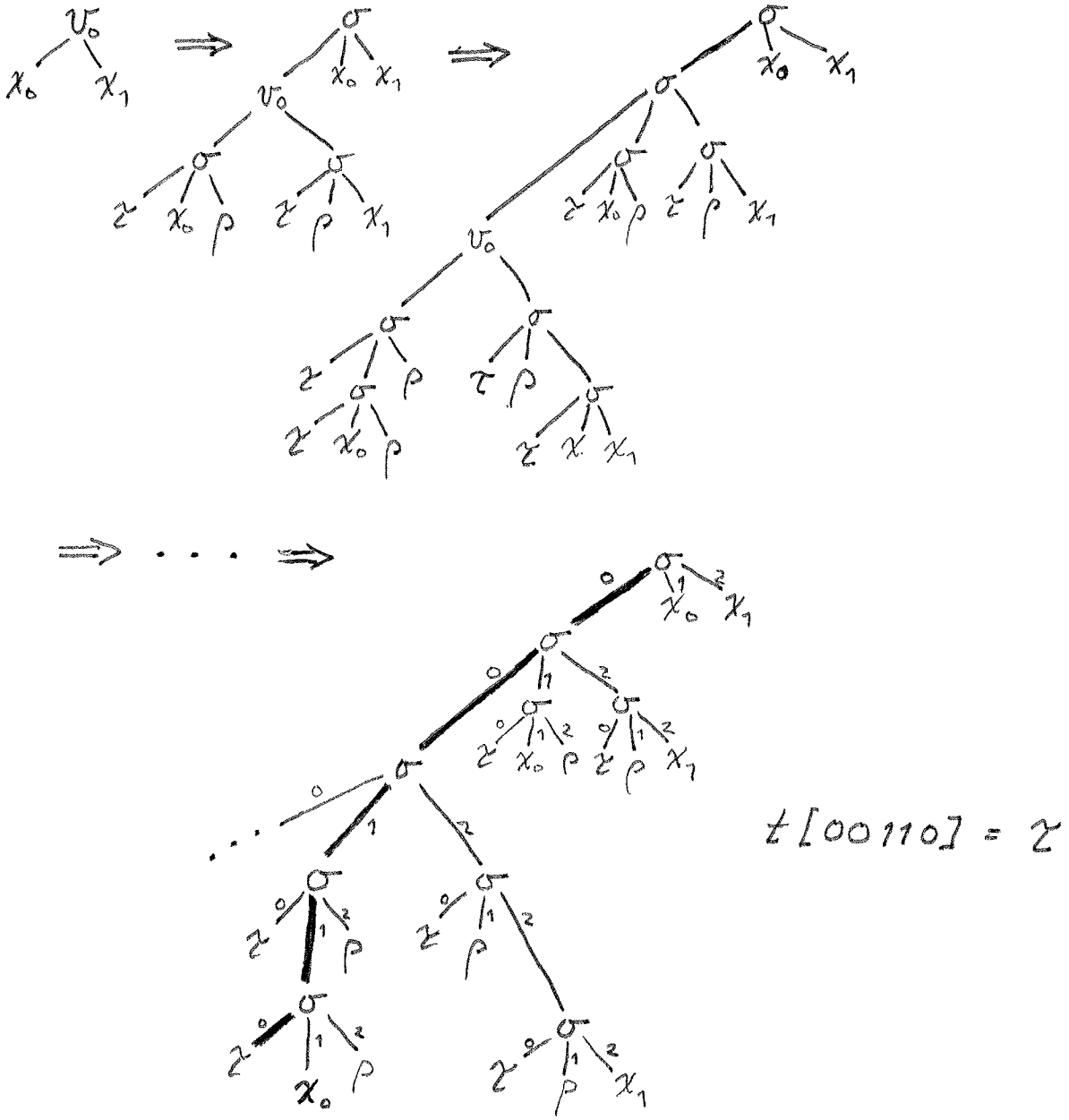
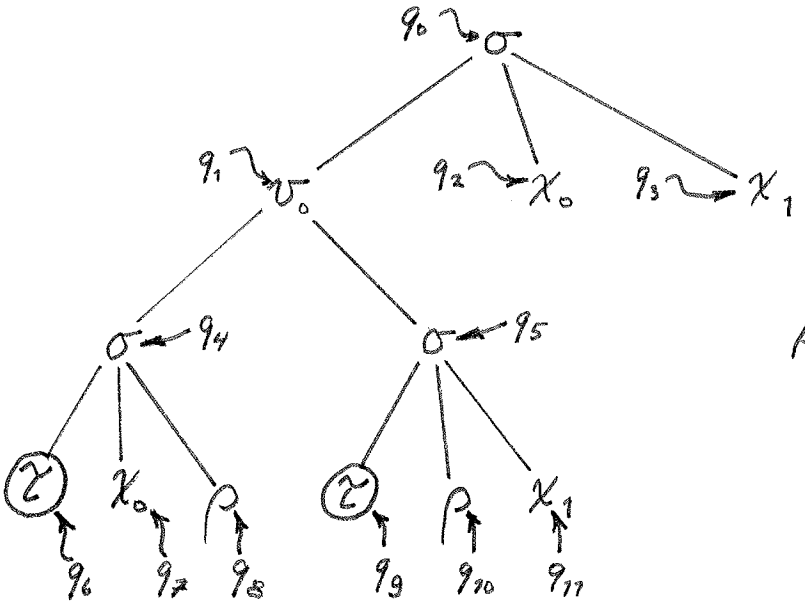


Figure 2a



Final states are circled

Figure 2b

States of P

State	Stack	Remaining Input	Move
q_0	Z_0	00110	i
q_1	Z_0	0110	ii
q_0	$q_1 Z_0$	0110	i
q_1	$q_1 Z_0$	110	ii
q_0	$q_1 q_1 Z_0$	110	i
q_2	$q_1 q_1 Z_0$	10	iii
q_4	$q_1 Z_0$	10	i
q_7	$q_1 Z_0$	0	iii
q_4	Z_0	0	i
q_6	Z_0	ε	accept

Figure 2c

Action of P on input 00110

$Q_W = \{q_0, q_1\}$

$Q_S = \{q_2, q_3, q_4\}$

$Q_e = \{p_0, p_1\}$

$F = \{q_5\}$

$\Sigma = \{0, 1\}$

$\Gamma = \{a, b\}$

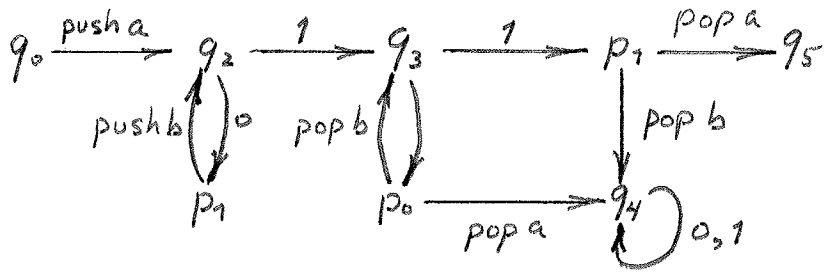


Figure 3a

A pushdown automaton in normal form.

f	0	1
q_2	q_1	q_3
q_3	p_0	p_1
q_4	q_4	q_4

g	p_0	p_1
q_0	q_4	q_5
q_1	q_3	q_4

h	
q_0	q_2
q_1	q_2

k∘f	0	1
q_2	$v_1(x_0, x_1)$	$v_3(x_0, x_1)$
q_3	x_0	x_1
q_4	$v_4(x_0, x_1)$	$v_4(x_0, x_1)$

k∘g	p_0	p_1
q_0	$v_4(x_0, x_1)$	$v_5(x_0, x_1)$
q_1	$v_3(x_0, x_1)$	$v_4(x_0, x_1)$

$$v_0(x_0, x_1) = v_2(v_4(x_0, x_1), v_5(x_0, x_1))$$

$$v_1(x_0, x_1) = v_2(v_3(x_0, x_1), v_4(x_0, x_1))$$

$$v_2(x_0, x_1) = \sigma(v_1(x_0, x_1), v_3(x_0, x_1))$$

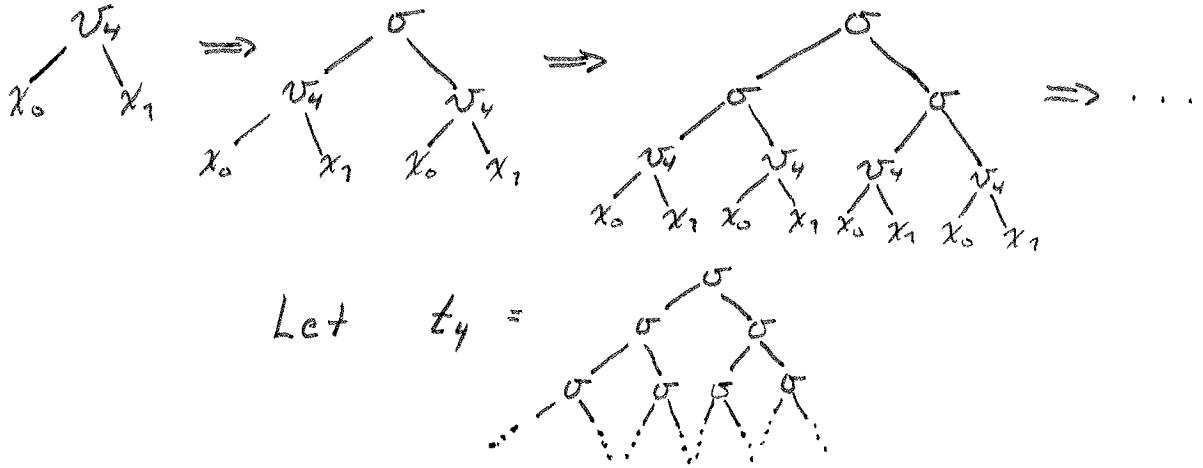
$$v_3(x_0, x_1) = \sigma(x_0, x_1)$$

$$v_4(x_0, x_1) = \sigma(v_4(x_0, x_1), v_4(x_0, x_1))$$

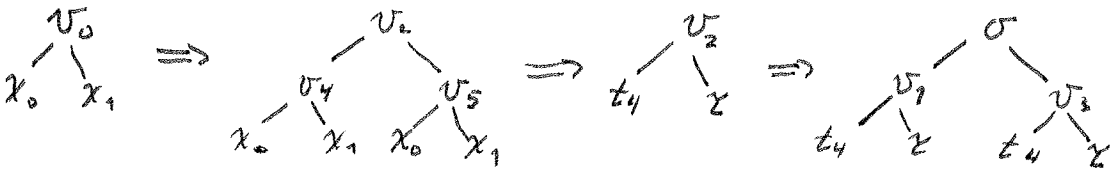
$$v_5(x_0, x_1) = z$$

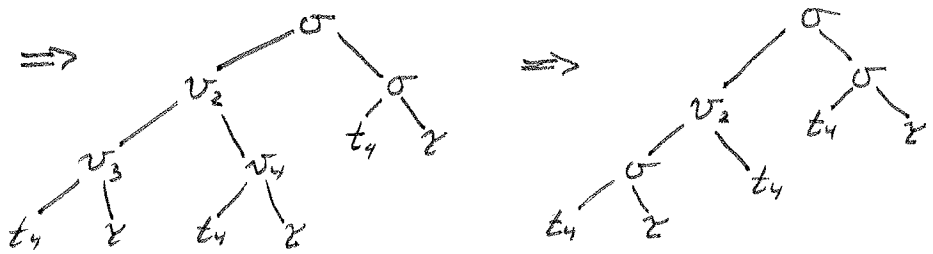
Figure 3b

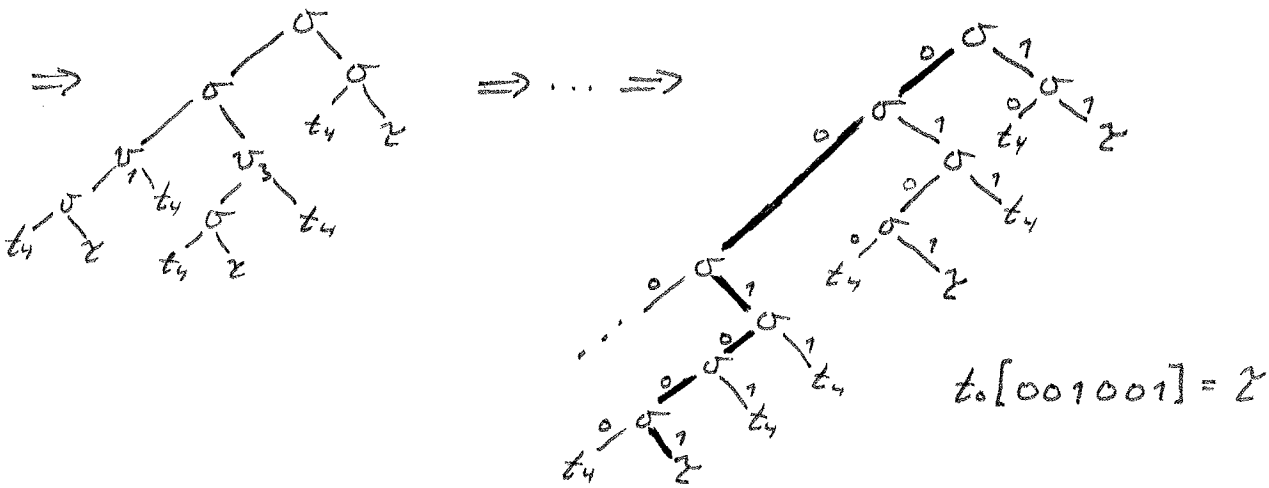
The type definition derived from 3a.



Let t_4 =







$t_0[001001] = z$

$t_0^{-1}[z] = \{0^n10^n1 \mid n \geq 0\}$

Figure 3c